Zu ADL gleichausdrucksstarke Basic-Action-Theorien im Situationskalkül

Studienarbeit von
Patrick Eyerich

Albert-Ludwigs-Universität Freiburg
Fakultät für Angewandte Wissenschaften
betreut von Prof. Dr. Bernhard Nebel

Zusammenfassung Aktionsformalismen wie z.B. GOLOG oder FLUX wurden unter dem Gesichtspunkt entwickelt, eine möglichst ausdrucksstarke Sprache zur Verfügung zu haben. Im Gegensatz dazu stand bei der Entwicklung von Planungsmethoden die Effizienz im Vordergrund, weshalb die Ausdrucksmächtigkeit der zunächst verwendeten Sprachen (allen voran STRIPS) eher gering war. Die Plansprache PDDL jedoch, welche im Jahre 1998 entwickelt wurde und ebenfalls auf STRIPS basiert, wurde im Laufe der Zeit durch die Erweiterung um immer mehr Funktionalität immer ausdrucksstärker, so dass es sehr interessant erscheint, ihre Ausdrucksstärke mit der von Aktionsformalismen zu vergleichen. Basic-Action-Theorien im Situationskalkül dienen dazu, eine dynamische Umgebung zu charakterisieren und sind die Grundlage von GOLOG. Wir definieren hier eine Menge von Basic-Action-Theorien, die über die gleiche Ausdrucksstärke verfügen, wie das ADL-Fragement von PDDL. Um diese Gleichheit zu zeigen, benutzen wir den Kompilationsansatz, der sich als sehr produktiv erwiesen hat, um die Ausdrucksmächtigkeit verschiedener Plansprachen miteinander zu vergleichen.

1 Einleitung

Während Aktionsformalismen und Planungsmethoden gemeinsame Wurzeln haben, insbesondere durch die Arbeiten zu Planen im Situationskalkül von C. Green [4], hat sich seit der Einführung von STRIPS [5] die Forschung auf diesen Gebieten in den vergangenen 30 Jahren weitgehend unabhängig voneinander entwickelt. Während bei den Aktionsformalismen von einer möglichst ausdrucksstarken Sprache ausgegangen wurde und Effizienzbetrachtungen eher zurück gestellt wurden, stand bei Planungsmethoden die Effizienz im Vordergrund und man beschränkte sich lange Zeit auf sehr einfache Plansprachen, die im Wesentlichen dem STRIPS-Ansatz folgten. Allerdings kann man seit geraumer Zeit eine gewisse Konvergenz feststellen. Dies lässt sich insbesondere an der Entwicklung der Plansprache PDDL und ihren Erweiterungen [7, 1] ablesen, wo unter anderem bedingte Effekte, Zeit und kontinuierliche Effekte behandelt werden.
Um Aktionsformalismen wie GOLOG [9] mit Planungssprachen vergleichbar zu machen, ist es hilfreich, die Ausdruckskraft verschiedener Ansätze miteinander

zu vergleichen. Eine Vorgehensweise, die sich dabei als sehr produktiv erwiesen hat, ist der Kompilationsansatz [3].
Mit dieser Arbeit wird ein erster Schritt in diese Richtung gemacht, indem Basic-Action-Theorien im Situationskalkül, auf dem GOLOG basiert, so eingeschränkt werden, dass sie über die gleiche Ausdrucksstärke wie das ADL-Fragment von PDDL verfügen. Diese Gleichheit wird gezeigt, indem Kompilationsschemata sowohl von eingeschränkten Basic-Action-Theorien nach ADL als auch in die andere Richtung angegeben werden.
Die weitere Arbeit ist folgendermaßen gegliedert: In Kapitel 2 stellen wir kurz PDDL im Allgemeinen und das für uns relevante ADL-Fragment mit seiner Syntax und Semantik genauer vor. In Kapitel 3 beschäftigen wir uns mit dem Situationskalkül und seinen Basic-Action-Theorien. Wir definieren neue *eingeschränkte* Basic-Action-Theorien und geben dann in Kapitel 4 Kompilationsschemata für beide Richtungen an, bevor wir in Kapitel 5 die erzielten Ergebnisse zusammenfassen und einen Ausblick auf weitere interessante, auf dieser Arbeit aufbauende, Fragestellungen geben.

2 PDDL

PDDL wurde 1998 von Drew McDermot entworfen [7] und seitdem kontinuierlich weiterentwickelt. Mittlerweile existiert Version 3.0 [10], welche im Rahmen der International Planning Competition 2006 eingesetzt werden wird. PDDL basiert auf einer Standardisierung des STRIPS-Formalismus, verfügt jedoch auch über zahlreiche weitere Konstrukte wie z.B. numerische Fluents, zeitabhängige Aktionen oder abgeleitete Prädikate.
Ein Planungsproblem wird zur Formalisierung in PDDL aufgeteilt in eine Domänenbeschreibung, die - z.B. durch parametrisierte Aktionen - das Verhalten einer Domäne beschreibt, sowie eine Problembeschreibung, welche spezifische Objekte, den Startzustand, das Ziel, sowie eine Metrik, mit der die Qualität von Plänen ermittelt werden kann, enthält.
Im Folgenden beschränken wir uns auf PDDL 2.1 Stufe 1 [1], welches im Wesentlichen dem ADL-Fragment [6] der Sprache entspricht.

2.1 EBNF des ADL-Fragmentes von PDDL 2.1

Zur Darstellung der Syntax des für das ADL-Fragment von PDDL 2.1 relevanten Teiles verwenden wir die hierzu übliche Extended BNF (EBNF) [7, 1]. Diese EBNF verwendet folgende Konventionen:

1. Jede Regel ist von der Form `<syntactic element> ::= expansion`.
2. Zwischen `<` und `>` stehen Namen von syntaktischen Elementen.
3. Was zwischen `[` und `]` steht ist optional.
4. * bedeutet „null mal oder öfter“, + bedeutet „einmal oder öfter“
5. Manche syntaktischen Elemente sind parametrisiert. Hat man z.B. `<list (x)> ::= (x*)` und eine Definition für `<symbol>` gegeben, so expandiert `<list(symbol)>` zu `(<symbol>*)`

6. Normale Klammern (und) sind ein Teil der zu definierenden Syntax und haben keine Bedeutung in der EBNF.

```
<domain>                    ::= (define (domain <name>)
                                        (:requirements :adl)
                                        [<types-def>]
                                        [<constantes-def>]
                                        [<predicates-def>]
                                        <action-def>*)
<types-def>                 ::= (:types <typed list1 (name)>)
<predicates-def             ::= (:predicates <atomic formula skeleton>+)
<atomic formula skeleton>   ::= (<predicate> <typed list2(variable)>)
<predicate>                 ::= <name>
<variable>                  ::= <name>

<typed list1 (x)>           ::= x*
<typed list1 (x)            ::= x+ - <primitive-type> <typed-list1(x)>
<primitive-type>            ::= <name>

<typed list2 (x)>           ::= x*
<typed list2 (x)            ::= x+ - <type2> <typed-list2(x)>
<primitive-type>            ::= <name>
<type2>                     ::= (either <primitive-type>+)
<type2>                     ::= <primitive-type>

<action-def>                ::= (:action <name>
                                    :parameters (<typed list2(variable)>)
                                    [:precondition <GD>]
                                    [:effect <effect>]

<GD>                        ::= ()
<GD>                        ::= <atomic formula(term)>
<GD>                        ::= <literal(term)>
<GD>                        ::= (and <GD>*)
<GD>                        ::= (or <GD>*)
<GD>                        ::= (not <GD>)
<GD>                        ::= (imply <GD> <GD>)
<GD>                        ::= (exists (typed list2(variable)>*) <GD>)
<GD>                        ::= (forall (typed list2(variable)>*) <GD>)
<literal(t)>                ::= <atomic formula(t)>
<literal(t)                 ::= (not <atomic formula(t)>)
<atomic formula(t)>         ::= (predicates> t*)
<term>                      ::= <name>
<term>                      ::= <variable>

<effect>                    ::= ()
```

```
<effect>                  ::= (and <c-effect>*)
<effect>                  ::= <c-effect>
<c-effect>                ::= (forall (<variable>*) <effect>)
<c-effect>                ::= (when <GD> <cond-effect>)
<c-effect>                ::= <p-effect>
<p-effect>                ::= (not <atomic formula(term)>)
<p-effect>                ::= <atomic formula(term)>
<cond-effect>             ::= (and <p-effect>*)
<cond-effect>             ::= <p-effect>

<problem>                 ::= (define (problem <name>)
                                  (:domain <name>)
                                  [<object declaration>]
                                  <init>
                                  <goal>
<object declaration>      ::= (:objects <typed list1(name)>)
<init>                    ::= (:init <literal(name)>*)
<goal>                    ::= (:goal <GD>)
```

In der durch die Original-Definition von PDDL 2.1 definierten Syntax [1] ist es möglich, in der Definition der Typen (`(:types ...)`) und der Deklaration der Objekte (`(:objects ...)`) zusammengesetzte Typen der Form `(either` $\texttt{T}_1$ $\texttt{T}_2$ `...)` zu verwenden. Hierdurch lassen sich Planungsprobleme definieren, die über keine vollständige Theorie verfügen, was wir aber von einem ADL-Planungsproblem erwarten.

Beispiel 1. Durch die Ausdrücke `(types T1 - (either T2 T3))` und `(:objects X1 - T1)` wird ein Typ `T1` deklariert, der ein Untertyp der Vereinigungsmenge von `T2` und `T3` ist; außerdem wird ein Objekt `X1` deklariert, dass vom Typ `T1` ist. Nun lässt sich aber nichts darüber aussagen, ob `X1` dem Typ `T2`, dem Typ `T3`, oder beiden zugehörig ist.
Gibt es z.B. ein Aktionsschema `(:action A :parameters z - T2 ...)`, so kann man nicht entscheiden, ob `A` mit Parameter `X1` angewendet werden kann.

Aus diesem Grund haben wir anstatt des syntaktischen Elements `<typed list (x)>` die beiden Elemente `<typed list1 (x)>` und `<typed list2 (x)>` eingeführt. `<typed list1 (x)>` erlaubt nur primitive Typen auf der rechten Seite und ersetzt an den kritischen Stellen `<typed list (x)>`, während `<typed list2 (x)>` dem herkömmlichen `<typed list (x)>` entspricht und an allen anderen Stellen auftaucht.

Offensichtlich handelt es sich bei Formeln, die ausgehend von `<GD>` gebildet wurden, um prädikatenlogische Formeln, wobei als Terme nur nullstellige Funktionen (Konstanten) zugelassen sind. Typen werden dabei als einstellige Prädikate aufgefasst und die Verwendung einer getypten Liste innerhalb eines Quantors als Kurzschreibweise für eine Implikation. Dies führt zu folgender Proposition:

Proposition 1. *Eine Formel, die ausgehend von* `<GD>` *gemäß der EBNF des ADL-Fragmentes von PDDL gebildet wurde, kann in der herkömmlichen Syntax der Prädikatenlogik geschrieben werden. Typen werden dabei zu einstelligen Prädikaten, getypte Listen zu Implikationen.*
Eine prädikatenlogische Formel, in der funktionale Terme nur in Form von Funktionen der Stelligkeit 0 *auftreten, kann als eine PDDL-Formel geschrieben werden, die auch durch Anwendung der Regeln der EBNF des ADL-Fragmentes von PDDL gebildet werden kann, wenn man mit* `<GD>` *startet.*

Definition 1 (Syntaxumformung prädikatenlogischer Formeln). *Sei* Π_1 *eine prädikatenlogische Formel in herkömmlicher Syntax. Mit* $\Xi^{PDDL}(\Pi_1)$ *bezeichnen wir dieselbe Formel in PDDL-Syntax, wobei bei jedem Fluenten das Situationsargument gestrichen wird.*
Sei Π_2 *eine Formel in PDDL-Syntax. Mit* $\Xi^{FOL}(\Pi_2)$ *bezeichnen wir die selbe Formel in herkömmlicher prädikatenlogischer Syntax, wobei jeder Fluent um ein Situationsargument als letzes Argument ergänzt wird. Getypte Listen werden dabei zu Implikationen mit einstelligen Prädikaten ohne Situationsargument.*

Beispiel 2. Sei $\Pi =$

```
(imply (exists (y)
               (or (forall (x - T) (and (P x y) (G y x)))
                   (F y)))
       (Z x y))
```

Es ist $\Xi^{FOL}(\Pi) =$

$$(\exists y(F(y) \vee (\forall x(T(x) \supset (P(x,y,s) \wedge G(y,x,s))))) \supset Z(x,y,s)$$

und $\Xi^{PDDL}(\Xi^{FOL}(\Pi)) =$

```
(imply (exists (y)
               (or (forall (x) (imply (T x) (and (P x y) (G y x))))
                   (F y)))
       (Z x y))
```

Man beachte hierbei, dass Π und $\Xi^{PDDL}(\Xi^{FOL}(\Pi))$ nahezu syntaktisch gleich sind. Der einzige Unterschied liegt darin, dass in Π `T` ein Typ ist und in $\Xi^{PDDL}(\Xi^{FOL}(\Pi))$ ein einstelliges Prädikat. Interpretiert man Typen als einstellige Prädikate, was wir im Folgenden tun wollen, so gibt es also keine semantischen Unterschiede.

Außerdem fordern wir, dass ein ADL-Planungsproblem folgenden Bedingungen genügt:

1. Es gibt keine Zyklen im Typ-Graphen.
2. Alle im ADL-Planungsproblem vorkommenden Objekte werden in der Klausel `(:objects ...)` genau einmal deklariert.
3. Alle im ADL-Planungsproblem vorkommenden Prädikate werden in der Klausel `(:predicates ...)` genau einmal deklariert.

4. Alle im ADL-Planungsproblem vorkommenden Typen werden in der Klausel `(:types ...)` genau einmal deklariert.

Betrachten wir die EBNF genauer, so sehen wir, dass jedes ADL-Planungsproblem P einem Grund-Schema genügt, welches folgendermaßen aussieht:

```
(define (domain DOMAIN)
(:requirements :adl)
(:types P_Types)
(:predicates P_Predicates)
P_Actions
)

(define (problem PROBLEM)
(:domain DOMAIN)
(:objects P_Objects)
(:init P_Init)
(:goal P_Goal)
)
```

Eine Probleminstanz P lässt sich also parametrisieren als Tupel

$$P = \langle P', P_{Objects}, P_{Init}, P_{Goal} \rangle$$

mit $P' = \langle P_{Types}, P_{Predicates}, P_{Actions} \rangle$.

2.2 Semantik von ADL

Wir verwenden hier die übliche Zustandsübergangssemantik von PDDL. Dieser liegen die folgenden Annahmen zu Grunde:

A1 Vollständige Theorie:
Es wird davon ausgegangen, dass in jedem Zustand die Wahrheitswerte aller Grundatome bekannt sind. Deshalb kann ein Zustand durch Angabe all seiner wahren Grundatome vollständig beschrieben werden. Grundatome, die nicht explizit als wahr klassifiziert werden, werden dann als falsch klassifiziert (dies ist die sogenannte Closed World Assumption (CWA)).

A2 Domain Closure Assumption (DCA):
Alle Objekte sind bekannt; ein Objekt, dessen Name nicht explizit eingeführt wird, existiert nicht.

A3 Unique Names Assumption (UNA):
Kein Objekt verfügt über zwei verschiedene Namen.

A4 Es gibt endlich viele Objekte und Aktionen.

Für PDDL wurde erstmals für Version 2.1 eine formale Semantik eingeführt [1], welche im Wesentlichen auf der STRIPS-Semantik von Lifschitz [8] aufbaute. In diese wurden die Erweiterungen wie z.B. Zeit, Gleichzeitigkeit und numerische

Variablen direkt eingebaut. Für diese Arbeit genügt uns jedoch der ADL-Teil, deshalb geben wir im Folgenden eine Semantik an, die nur die für uns relevanten Stellen berücksichtigt.
Die Semantik beruht auf dem bekannten Zustandsübergangs-Modell.

Definition 2 (Planungsinstanz). *Eine Planungsinstanz ist ein Paar*

$$\text{I}=(\text{Dom},\text{Prob})$$

wobei Dom=(Rs,As,arity) *ein Tupel ist bestehend aus Relationssymbolen* Rs*, Aktionen* As *und einer Funktion* arity*, die alle Symbole auf ihre Stelligkeit abbildet.* Prob=(Os,Init,G) *ist ein Tripel bestehend aus den Objekten der Domäne, der Initialzustand-Spezifikation und der Zielzustand-Spezifikation.*
Die Grundatome Atms *einer Planungsinstanz sind die endlich vielen Ausdrücke, die entstehen, wenn man die Relationssymbole* Rs *auf die Objekte* Os *anwendet (wobei man natürlich auf die Stelligkeit achten muss).*
Init *ist eine Menge von Literalen, die aus Grundatomen aus* Atms *besteht. Die Zielbedingung* G *ist ein Satz, der Grundatome enthalten kann.*
As *ist eine Menge von Aktions-Schemata in PDDL-Syntax. Die Atom-Schemata, die in diesen Aktions-Schemata benutzt werden, entstehen durch Anwendung der Relations-Symbole, die in der Domäne definiert wurden, auf die Objekte in* Os *und Schema-Variablen.*

Wir wollen im Folgenden instanziierte Aktions-Schemata als Zustandsübergänge interpretieren. Dazu definieren wir als nächstes, was wir unter einem Zustand verstehen.

Definition 3 (Zustand). Atms_I *sei die Menge von Grundatomen einer Planungsinstanz* I*. Ein Zustand* z *ist eine Teilmenge von* Atms(I)*.*

Wir benutzen hier gemäß Annahme A1 die Closed World Assumption. Als nächstes werden wir Aktions-Schemata instanziieren.

Definition 4 (Abgeflachte Aktionen). *Sei eine Planungsinstanz* I *mit einem Aktions-Schema* $\text{A}\in\text{As}_I$ *gegeben. Die Menge der abgeflachten Aktions-Schemata* flatten(A) *ist definiert als die Menge* S*, die mit* A *initialisiert wird und folgendermaßen konstruiert wird:*

1. *Solange* S *ein Aktions-Schema* X *enthält, dass einen konditionalen Effekt* `(when P Q)` *enthält, erzeugen wir zwei neue Schemata* X' *und* X"*, welche Kopien von* S *ohne den konditionalen Effekt sind. Die eine Kopie erhält* `P` *als zusätzliche Vorbedingung und* Q *als zusätzlichen Effekt, die andere erhält* `(not P)` *als zusätzliche Vorbedingung. Dann streichen wir* X *aus* S *und ergänzen* S *um* X' *und* X"*.*
2. *Solange* S *ein Aktions-Schema* X *enthält, dass eine Formel mit einem All-Quantifizierer* `(forall (var`$_1$`...var`$_k$`) P)` *enthält, ersetzen wir* X *durch ein Aktions-Schema* X'*, dass bis auf den Quantifizierer eine Kopie von* X *ist, in der aber* `(forall (var`$_1$`...var`$_k$`) P)` *ersetzt wurde durch die Konjunktion der Sätze, die gebildet wurden durch Substitution aller passenden Objekte in* I *für jede Variable* `var`$_1$`...var`$_k$ *in* `P`*.*

3. *Solange* S *ein Aktions-Schema* X *enthält, dass eine Formel mit einem Existenz-Quantifizierer* `(exists (var`$_1$`...var`$_k$`) P)` *enthält, ersetzen wir* X *durch ein Aktions-Schema* X', *dass bis auf den Quantifizierer eine Kopie von* X *ist, in der aber* `(exists (var`$_1$`...var`$_k$`) P)` *ersetzt wurde durch die Disjunktion der Sätze, die gebildet wurden durch Substitution aller passenden Objekte in* I *für jede Variable* `var`$_1$`...var`$_k$ *in* `P`.

Diese Schritte werden so lange wiederholt, bis keiner mehr anwendbar ist.

Nachdem alle Aktions-Schemata abgeflacht wurden, kann man sie durch die übliche Substitution durch Objekte für Parameter instanziieren:

Definition 5 (Instanziierte Aktionen). *Sei eine Planungsinstanz* I *mit einem Aktions-Schema* A∈As$_I$ *gegeben. Die Menge der instanziierten Aktionen* GA$_A$ *für* A *ist definiert als die Menge aller Stukturen* a*, die durch Substitution aller Schema-Variablen durch Objekte in allen Schemata X∈flatten(A) entsteht. Die Komponenten von a sind:*

1. Name$_a$ *ist der Name des Aktions-Schemas* X *zusammen mit den Werten, die für die Parameter von* X *substituiert wurden.*
2. Pre$_a$ *ist die propositionale Vorbedingung von a. Die Menge der Grundatome, die in Pre$_a$ vorkommen, wird mit* GPre$_a$ *bezeichnet.*
3. Add$_a$*, die positive Nachbedingung von a, ist die Menge von Grundatomen, die im Effekt von a wahr gesetzt werden.*
4. Del$_a$*, die negative Nachbedingung von a, ist die Menge von Grundatomen, die im Effekt von a falsch gesetzt werden.*

Definition 6 (Anwendbare Aktion). *Sei a eine instanziierte Aktion. a ist anwendbar im Zustand z, wenn gilt* $z \models Pre_a$.

Definition 7 (Plan in ADL). *Ein Plan P ist eine Sequenz von Aktionen* $[a_1...a_k]$*, die ausführbar und gültig ist. Eine Sequenz von Aktionen ist ausführbar, wenn es eine Sequenz von Zuständen* $\{z_i\}_{i=0...k+1}$ *gibt, so dass* a_i *in* z_i *anwendbar ist,* z_0 *der Initial-Zustand der Planungsinstanz ist und für jedes* $i = 0...k$ *gilt, dass* z_{i+1} *aus* z_i *entsteht, indem alle Grundatome aus* Add$_a$ *zu* z_i *hinzugefügt werden und alle Grundatome aus* Del$_a$ *aus* z_i *gelöscht werden.*
Eine Sequenz von Aktionen ist gültig, wenn gilt: $z_{i+1} \models G$.

3 Situationskalkül

GOLOG ist eine von Hector Levesque und anderen entwickelte Aktionssprache, die auf dem Situationskalkül basiert [9, 2]. Der Situationskalkül ist eine Sprache der ersten Stufe mit einigen Eigenschaften der zweiten Stufe und dient der Repräsentation sich dynamisch verändernder Systeme.
Eine Sequenz von Aktionen wird Situation genannt und durch einen Term der ersten Stufe beschrieben. Dabei ist s_0 die Initialsituation, also die Situation, in der noch keine Aktionen durchgeführt wurden. Desweiteren gibt es ein binäres

Funktionssymbol *do*. $do(\alpha, s)$ beschreibt die Situation, die entsteht, wenn man in der Situation s die Aktion α ausführt.
Relationen, deren Wahrheitswert von Situation zu Situation variiert, werden relationale Fluenten genannt. Sie haben neben ihren regulären Argumenten ein weiteres, dass per Konvention als letzes notiert wird. So bedeutet beispielsweise $At(x, y, s)$, dass sich Objekt x in Situation s an Position y befindet.
Desweiteren gibt es die binären Prädikatensymbole $Poss$ und $\sqsubset$. $Poss(a, s)$ gibt an, unter welchen Bedingungen die Aktion a in Situation s angewendet werden kann; $s \sqsubset s'$ definiert eine Ordnungsrelation auf Situationen (die als Aktions-Sequenzen interpretiert werden) und bedeutet, dass s eine echte Teilsequenz von s' ist.
Dynamische Umgebungen werden im Situationskalkül beschrieben, indem man für jede von einem Agenten direkt ausführbare Aktion (dies sind die sogenannten primitiven Aktionen) eine Vorbedingung angibt, für jeden Fluenten definiert, unter welchen Bedingungen er in einer Situation wahr ist, sowie den initialen Zustand der Welt beschreibt. Außerdem wird die Existenz von einigen allgemeinen Situationskalkül-Axiomen sowie Unique-Names-Axiomen vorausgesetzt. All diese Teile zusammen bilden dann eine sogenannte Basic-Action-Theorie (siehe weiter unten).
GOLOG bietet eine Reihe von komplexen Aktionen, dazu gehören Test-Aktionen, Sequenzen, nicht-deterministische Auswahl von Aktionen, nicht-deterministische Auswahl von Aktions-Argumenten sowie nicht-deterministische Iteration. Außerdem lassen sich Prozeduren definieren. Bei all diesen Konstrukten handelt es sich um Makros. Eine Sequenz von Aktionen, die ein solches Makro enthält, kann im Allgemeinen für mehrere Sequenzen von primitiven Aktionen stehen, die zudem sehr viel länger sein können. Bei der Ausführung eines Planes durch einen Agenten können jedoch natürlich nur direkt die primitiven Aktionen ausgeführt werden. Da die Länge eines Planes eine nicht unerhebliche Rolle bei der Bewertung einer Kompilation spielt, betrachten wir in dieser Arbeit nur primitive Aktionen.

Anmerkung 1 (uniforme Formeln). Im Folgenden sprechen wir von einer in σ uniformen Formel f, wenn in f weder $Poss$ noch $\sqsubset$ vorkommt, nicht über Situationsvariablen quantifiziert wird, Gleichheit nicht im Zusammenhang mit Situationen vorkommt und nur σ als Situationsterm in Fluenten vorkommt.

Wir definieren jetzt, was wir unter einem Plan in einer Basic-Action-Theorie verstehen [2, Seite 38]. Da eine Basic-Action-Theorie zwar eine Initialsituation aber keine Zielbedingung beschreibt, definieren wir einen Plan in einer Basic-Action-Theorie T in Abhängigkeit von einer Situationskalkülformel $G(s)$ mit einziger freier Variable s:

Definition 8 (Plan in einer Basic-Action-Theorie).
Ein Plan für eine Situtationskalkülformel $G(s)$ in T ist ein variabelfreier Situationsterm σ, für den gilt: $T \models executable(\sigma) \wedge G(\sigma)$. Dabei bedeutet $executable(s)$, dass alle Aktionen nacheinander ausführbar sind und ist folgendermaßen definiert: $executable(s) := (\forall a, s^*).do(a, s^*) \sqsubseteq s \supset Poss(a, s^*)$.

Ein variabelfreier Situationsterm ist im Wesentlichen eine Sequenz primitiver Aktionen ausgehend von s_0, die man auch schreiben kann, indem man die Aktionen „von rechts nach links“ auflistet. Zum Beispiel kann man statt $do(a_3, do(a_2, do(a_1, s_0)))$ kürzer und übersichtlicher $[a_1, a_2, a_3]$ schreiben.

3.1 Basic-Action-Theorien

Im Folgenden seien alle ungebundenen Variablen allquantifiziert.
Eine Basic-Action-Theorie T besteht aus folgenden Teilen [2, Seite 58]:

$$T = \Sigma \cup T_{SSA} \cup T_{PA} \cup T_{UNA} \cup T_{s_0}$$

1. Σ sind die grundlegenden Axiome des Situationskalküls:
   $\Sigma_1 : do(a_1, s_1) = do(a_2, s_2) \supset a_1 = a_2 \wedge s_1 = s_2$
   $\Sigma_2 : (\forall P).P(s_0) \wedge (\forall a, s)[P(s) \supset P(do(a, s))] \supset (\forall s)P(s)$
   $\Sigma_3 : \neg s \sqsubset s_0$
   $\Sigma_4 : s \sqsubset do(a, s') \equiv s \sqsubseteq s'$
2. T_{SSA} ist eine Menge von Successor State Axiomen für funktionale und relationale Fluenten.
   Ein Successor State Axiom für einen $n+1$-stelligen relationalen Fluenten F ist ein Satz der Form:

   $$F(x_1, ..., x_n, do(a, s)) \equiv \Phi_F(x_1, ..., x_n, a, s)$$

   wobei $\Phi_F(x_1, ..., x_n, a, s)$ eine in s uniforme Formel mit freien Variablen $a, s, x_1, ..., x_n$ ist.
   Ein Successor State Axiom für einen $n+1$-stelligen funktionalen Fluenten f ist ein Satz der Form:

   $$f(x_1, ..., x_n, do(a, s)) = y \equiv \phi_f(x_1, ..., x_n, y, a, s)$$

   wobei $\phi_f(x_1, ..., x_n, y, a, s)$ eine in s uniforme Formel mit freien Variablen $a, s, y, x_1, ..., x_n$ ist.
3. T_{PA} ist eine Menge von Action Precondition Axiomen. Ein Action Precondition Axiom ist ein Satz der Form

   $$Poss(A(x_1, ...x_n), s) \equiv \Pi_A(x_1, ..., x_n, s)$$

   wobei A ein n-stelliges Aktionssymbol ist und $\Pi_A(x_1, ..., x_n, s)$ eine in s uniforme Formel mit freien Variablen $s, x_1, ..., x_n$.
4. T_{UNA} ist die Menge der Unique Name Axiome für Aktionen. Diese haben die Form $A(\boldsymbol{x}) \neq B(\boldsymbol{y})$ für je zwei unterschiedliche Aktionsnamen A und B; sowie $A(x_1, ..., x_n) = A(y_1, ..., y_n) \supset x_1 = y_1 \wedge ... \wedge x_n = y_n$ für jeden Aktionsnamen A.
5. T_{s_0} ist eine Menge von in s_0 uniformen Sätzen der ersten Stufe. T_{s_0} wird oft die Initial Database genannt. Im Allgemeinen enthält T_{s_0} auch situationsunabhängige Sätze.

Wir gehen davon aus, dass für jede Aktion genau ein Action-Precondition-Axiom und für jeden Fluenten genau ein Successor-State-Axiom existiert. Zusätzlich wird folgende Konsistenz-Eigenschaft für funktionale Fluenten (*functional fluent consistency property*) gefordert:
Angenommen f ist ein funktionaler Fluent, dessen Successor State Axiom in T_{SSA} folgendermaßen lautet:

$$f(\boldsymbol{x}, do(a, s)) = y \equiv \phi_f(\boldsymbol{x}, y, a, s)$$

Dann gilt:

$$\begin{aligned} T_{UNA} \cup T_{s_0} \models\ & (\forall \boldsymbol{x}).(\exists y)\phi_f(\boldsymbol{x}, y, a, s) \\ & \wedge[(\forall y, y').\phi_f(\boldsymbol{x}, y, a, s) \wedge \phi_f(\boldsymbol{x}, y', a, s) \supset y = y']. \end{aligned}$$

Diese Bedingung besagt, dass ϕ_f für alle möglichen Parameter $\boldsymbol{x}$ von f einen Wert festlegt und dass dieser Wert eindeutig ist.

Auf den ersten Blick wird klar, dass eine Basic-Action-Theorie in ihrer allgemeinen Form ein mächtigeres Konstrukt als das ADL-Fragment von PDDL ist (z.B. haben wir hier keine vollständige Theorie und es gibt potentiell unendlich viele Objekte). Nun stellt sich natürlich die Frage, ob sich für eine sinnvolle Teilmenge aller Basic-Action-Theorien zeigen lässt, dass diese gleichmächtig zu ADL sind. Wir geben im Folgenden einige Einschränkungen an, die eine solche Teilmenge definieren, und zeigen, dass sich alle Probleme, die diesen Einschränkungen genügen, nach ADL kompilieren lassen. Dass wir nicht zu streng vorgegangen sind, zeigen wir, indem wir auch ein Kompilationsschema für die andere Richtung angeben.

3.2 Eingeschränkte Basic-Action-Theorien

Wir nennen im Folgenden eine Basic-Action-Theorie T *eingeschränkt*, falls sie folgende Bedingungen erfüllt:

E1 Es gibt keine Funktionsterme mit Ausnahme von nullstelligen Funktionen (Konstanten). Aufgrund dieser Bedingung können wir auf die oben erwähnte Konsistenz-Eigenschaft für funktionale Fluenten verzichten.

E2 Alle Successor-State- Axiome sind in einer speziellen Form. Das SSA eines Fluenten F sieht folgendermaßen aus:

$$F(x_1, ..., x_n, do(a, s)) \equiv \bigvee_{l=1}^{n} \phi_l \tag{1}$$

mit endlichem n. Genau ein ϕ_l ist von der Form:

$$\begin{aligned} F(x_1, ..., x_n, s) \wedge \neg\ & ((\exists ...)([\varphi_1 \wedge](a = A_1(y_{11}, ..., y_{1m_1}))) \\ & \vee(\exists ...)([\varphi_2 \wedge](a = A_2(y_{21}, ..., y_{2m_2}))) \\ & \vee(\exists ...)([\varphi_3 \wedge](a = A_3(y_{31}, ..., y_{3m_3}))) \\ & \vee ..) \end{aligned} \tag{2}$$

alle anderen ϕ_l sind von der Form

$$(\exists ...)([\varphi_l \wedge](a = A(y_1, ..., y_m))) \tag{3}$$

wobei jeweils existenzquantifiziert wird genau über die Variablen y_i, für die $y_i \neq x_j$ für alle j; über diejenigen also, die als Parameter der Aktion, aber nicht als Argument des Fluenten auftreten.
Was innerhalb eckiger Klammern steht, ist optional. Die φ_l sind prädikatenlogische Formeln ohne funktionale Terme und verfügen über keine freien Variablen.
Jedes SSA *muss* genau einen Term der Form (2) haben. Außerdem darf natürlich jede Aktion A nur einmal in einem SSA vorkommen.
Der Fluent F wird in Situation $do(a, s)$ also genau dann wahr, wenn einer der Terme ϕ_l wahr ist. Jeder Term ϕ_l ist nun von der Form (3) oder (2).
Um zu beschreiben, wie F wahr werden kann, wird Form (3) verwendet. F ist dabei in Situation $do(a, s)$ wahr, wenn es sich bei a um die Aktion A gehandelt hat. Optional lassen sich noch Bedingungen in Form eines φ_l angeben, die zusätzlich erfüllt sein müssen, damit F wahr wird.
Form (2) beschreibt, dass F in $do(a, s)$ wahr ist, falls es in s wahr war und die ausgeführte Aktion keine der Aktionen A_i ist. Auch hier können optional noch zusätzliche Bedingungen durch die φ_l gestellt werden. Dann ist F in $do(a, s)$ wahr, falls es in s wahr war und für jedes l entweder die ausgeführte Aktion keine der Aktionen A_l war, oder φ_l nicht erfüllt ist (oder natürlich beides).
Die φ_l entsprechen auf der ADL-Seite bedingten Effekten.
Die Forderung, dass jedes SSA genau einen Term der Form (2) haben muss, machen wir deshalb, weil im ADL-Fragment von PDDL Prädikate ihren Wert nur durch eine Aktion ändern können. Ist ein Fluent also wahr, muss er solange wahr bleiben, bis er durch eine Aktion wieder falsch gemacht wird. Falls es keine Aktion gibt, die einen Fluenten wieder falsch macht, so muss er immer wahr bleiben.

Beispiel 3. Wir betrachten eine Domäne, in der Lastautos und Menschen existieren. Das Fluent $At(x, y, s)$ soll angeben, dass x sich in Situation s an Position y befindet, wobei x ein Lastauto oder ein Mensch sein kann. Im Folgenden gehen wir davon aus, dass $In(x, y, s)$ bereits definiert wurde und bedeutet, dass sich der Mensch x im Lastauto y befindet.
Außerdem gehen wir davon aus, dass es eine Aktion $Fahren(l, v, n)$ gibt, die beschreibt, dass das Lastauto l von v nach n fährt.
Das SSA von $At(x, y, s)$ kann folgendermaßen definiert werden:

$$\begin{aligned} At(x, y, do(a, s)) \equiv\ & (\exists v)(a = Fahren(x, v, y)) \vee \\ & (\exists l, v)(In(x, l, s) \wedge a = Fahren(l, v, y)) \vee \\ & At(x, y, s) \wedge \neg(((\exists n)(a = Fahren(x, y, n))) \\ & \quad \vee((\exists l, n)(In(x, l, s) \wedge a = Fahren(l, y, n)))) \end{aligned}$$

Hier sind das erste und zweite Disjunkt von der Form (3) und das dritte von der Form (2).

E3 T_{s_0} besteht *nur* aus den folgenden Sätzen:

1.) Es gibt für jeden relationalen Fluenten F mit $n+1$ Argumenten einen Satz der Form

$$F(\boldsymbol{x}, s_0) \equiv x_1 = d_{11} \wedge ... \wedge x_n = d_{1n} \vee ... \vee x_1 = d_{m1} \wedge ... \wedge x_n = d_{mn} \quad (4)$$

mit m endlich.

2.) Für jedes situationsunabhängige Prädikat P mit n Argumenten gibt es einen Satz der Form

$$P(\boldsymbol{x}) \equiv x_1 = d_{11} \wedge ... \wedge x_n = d_{1n} \vee ... \vee x_1 = d_{m1} \wedge ... \wedge x_n = d_{mn} \quad (5)$$

mit m endlich.

Wir definieren also für jeden Fluenten und jedes situationsunabhängige Prädikat, für welche Kombinationen von Objekten sie in der Initialsituation s_0 wahr sind. Dies impliziert natürlich, dass sie für alle anderen Kombinationen von Objekten falsch sind, so dass wir in s_0 über eine vollständige Theorie verfügen.

3.) Es gibt ein Domain Closure Axiom:

$$\forall x(x = d_1 \vee ... \vee d_n) \quad (6)$$

indem jede in T vorkommende Konstante als ein d_i auftaucht.

E4 Es gibt Unique Names Axiome für Konstanten.

4 Kompilationsschemata

4.1 Einführung

Um die Ausdrucksstärke von verschiedenen Planungsformalismen zu untersuchen, entwickelte Nebel sogenannte Kompilationsschemata [3]. Dabei handelt es sich um ein formales Werkzeug zur Messung der relativen Ausdrucksstärke von Planungsformalismen. Die Intuition ist dabei, dass ein Formalismus X so mächtig wie ein Formalismus Y ist, falls sich alle Domänenbeschreibungen und Pläne von Y „einfach“ innerhalb von X ausdrücken lassen.
Kompilationsschemata sind lösungserhaltende Abbildungen polynomialer Größe von Y-Domänen in X-Domänen. Während wir hierbei die Größe des Resultates betrachten, spielt die für die Kompilation benötigte Zeit keine Rolle. Tatsächlich ist es für die Berechnung der Ausdruckskraft nicht wichtig, ob die Kompilation polynomiell berechenbar, exponentiell berechenbar, berechenbar oder sogar nicht-rekursiv ist.
Bei einem Kompilationsschema werden die Operatoren der Planungsinstanz von Y unabhängig von Start- und Zielzustand übersetzt. Wäre dies nicht der Fall,

könnte ein Kompilationsschema einfach das Plan-Existenz-Problem in Y lösen und dann eine sehr kleine, lösungsbewahrende Instanz in X erzeugen. Daraus ließe sich schließen, dass alle Planformalismen dieselbe Ausdruckskraft haben, was natürlich nicht Sinn der Sache ist.
Da es neben der Domänenstruktur und des von ihr benötigten Platzes auch sehr wichtig ist, wieviel Platz ein Plan braucht, wird zusätzlich zwischen Kompilationsschemata unterschieden, die die Plangröße *exakt*, *linear* oder *polynomiell* bewahren.
Hier wollen wir nicht die Ausdrucksstärke zweier Planungsformalismen miteinander vergleichen, sondern diejenige von zwei sich syntaktisch relativ fremden Formalismen. Deshalb haben wir die Definition von Kompilationsschemata leicht angepasst.

Im Folgenden bezeichnen wir eine Basic-Action-Theorie ohne T_{s_0} als BAT-Domäne $D = \langle \Sigma, T_{SSA}, T_{PA}, T_{UNA} \rangle$ und eine solche BAT-Domäne zusammen mit T_{s_0} und einer Situationskalkülformel $G(s)$ (Zielformel) mit einziger freier Variable s als BAT-Instanz $I = \langle D, T_{s_0}, G(s) \rangle$.

Definition 9 (Kompilationsschema BAT nach ADL). *Gegeben sei ein Tupel von Funktionen $\boldsymbol{f}$=$\langle f_{Types}, f_{Predicates}, f_{Actions}, f_{Objects}, f_{Init}, f_{Goal} \rangle$, dass eine Funktion F von einer BAT-Instanz $I = \langle D, T_{s_0}, G(s) \rangle$ auf ein ADL-Planungsproblem $F(I)$ wie folgt induziert:*

$$F(I) = \langle f_{Types}(T_{s_0}), f_{Predicates}(D), f_{Actions}(D), f_{Objects}(D, T_{s_0}), \\ f_{Init}(T_{s_0}), f_{Goal}(G(s)) \rangle$$

***f** ist ein Kompilationsschema, gdw.*

1. *Es gibt einen Plan für I, gdw. es einen Plan für $F(I)$ gibt.*
2. *Die Größe des Resultates von $f_{Types}, f_{Predicates}, f_{Actions}, f_{Objects}, f_{Init}$ und f_{Goal} ist polynomial in der Größe ihres Argumentes.*

Definition 10 (Kompilationsschema ADL nach BAT). *Gegeben sei ein Tupel von Funktionen $\boldsymbol{g}$=$\langle g_{\Sigma}, g_{T_{SSA}}, g_{T_{PA}}, g_{T_{UNA}}, g_{T_{s_0}}, g_{Goal} \rangle$, dass eine Funktion G von einem ADL-Planungsproblem $P = \langle P', P_{Init}, P_{Goal} \rangle$ auf eine BAT-Instanz $G(P)$ wie folgt induziert:*

$$G(P) = \langle g_{\Sigma}, g_{T_{SSA}}(P'), g_{T_{PA}}(P'), g_{T_{UNA}}(P'), \\ g_{T_{s_0}}(P_{Init}, P_{Objects}), g_{Goal}(P_{Goal}) \rangle$$

***g** ist ein Kompilationsschema, gdw.*

1. *Es gibt einen Plan für P, gdw. es einen Plan für $G(P)$ gibt.*
2. *Die Größe des Resultates von $g_{\Sigma}, g_{T_{SSA}}, g_{T_{PA}}, g_{T_{UNA}}, g_{T_{s_0}}$ und g_{Goal} ist polynomial in der Größe ihres Argumentes.*

Wenn nun ein Kompilationsschema von einer eingeschränkten Basic-Action-Theorie nach ADL existiert, so existiert für jede Problem-Instanz einer eingeschränkten Basic-Action-Theorie ein lösungsbewahrendes ADL-Planungsproblem. Es wäre also gezeigt, dass eingeschränkte Basic-Action-Theorien nicht

mächtiger sind als ADL-Planungsprobleme. Umgekehrt kann man, sollte ein Kompilationsschema für die andere Richtung existieren, jedes Planungsproblem in ADL auch in einer eingeschränkten BAT-Instanz forumlieren, womit gezeigt wäre, dass ADL nicht mächtiger ist, als es eingeschränkte Basic-Action-Theorien sind.
Existieren also Kompilationsschemata für beide Richtungen, so sind beide Formalismen gleich ausdrucksstark.
Der praktische Nutzen liegt darin begründet, dass man, wenn man während des Ablaufs eines GOLOG-Programms auf ein Planungsproblem stößt, dass in einer eingeschränkten Basic Action Theorie formuliert ist, dieses nach ADL übersetzen kann, es dort mit einem effektiven Planungssystem lösen und dann zurück in den Situationskalkül übersetzen kann. Voraussetzung dafür ist natürlich, dass die Kompilationsschemata in polynomieller Zeit berechenbar sind. In den Kapitel 4.2 und 4.4 geben wir Kompilationsschemata an, für die das der Fall ist.
Ein wesentlicher Unterschied zwischen PDDL und Situationskalkül ist die Notation von Veränderungen in der Domäne. Im Situationskalkül geschieht dies durch die Successor-State-Axiome. Ein solches SSA $F(\boldsymbol{x}, do(a, s)) \equiv \phi_F(x_1, ..., x_n, a, s)$ beschreibt den Zustand eines Fluenten F nach Ausführung einer Aktion a in Situation s. Dabei muss $\phi_F(x_1, ..., x_n, a, s)$ uniform in s sein, was intuitiv bedeutet, dass $\phi_F(x_1, ..., x_n, a, s)$ nur von der einen Situationsvariable s abhängt, also über die Markov Eigenschaft verfügt. Im Gegensatz dazu werden in PDDL Aktionen zusammen mit ihren Vorbedingungen, Parametern und Effekten kodiert. Veränderungen an der Domäne werden also innerhalb der Beschreibung einer Aktion notiert.
Bei einer Übersetzung von eingeschränkten Basic-Action-Theorien nach ADL gilt es also, eine Abbildung von der „Fluent-zentrierten“ Notation im Situationskalkül in die „Aktions-zentrierte“ in PDDL zu finden.

4.2 Ein Kompilationsschema von eingeschränkten BAT's nach ADL

Wann immer wir im Folgenden ein ADL-Prädikat P aus einem Situationskalkül-Fluenten F mit $n + 1$ Argumenten erzeugen, übergeben wir P n Argumente, nämlich genau diejenigen von F bis auf das Situationsargument.
Den leeren Ausdruck bezeichnen wir mit ϵ.

Wir erzeugen aus einem BAT-Problem $I = \langle D, T_{s_0}, G(s)\rangle$ ein ADL-Planungsproblem. Dazu geben wir die Funktionen $f_{Types}(T_{s_0})$, $f_{Predicates}(D)$, $f_{Actions}(D)$, $f_{Objects}(T_{s_0})$, $f_{Init}(T_{s_0})$ und $f_{Goal}(G(s))$ an. In Kapitel 4.3 zeigen wir dann, dass es sich bei **f** um ein Kompilationsschema handelt.
Die Resultate der sechs Funktionen setzen sich gemäß folgendem Schema zu einem ADL-Planungsproblem zusammen:

```
(define (domain DOMAIN)
(:requirements :adl)
(:types f_Types(T_s0))
(:predicates f_Predicates(D))
```

$f_{Actions}(D)$
)

(define (problem PROBLEM)
(:domain DOMAIN)
(:objects $f_{Objects}(D)$)
(:init $f_{Init}(T_{s_0})$)
(:goal $f_{Goal}(G(s))$)
)

1. $f_{Types}(T_{S_0})$

$$f_{Types}(T_{s_0}) = \epsilon$$

2. $f_{Objects}(D, T_{s_0})$
Die Funktion $f_{Objects}(D, T_{s_0})$ liefert einen PDDL-Ausdruck zurück, der sämtliche vorkommenden Objekte deklariert. Da wir bei der Kompilation keine getypten Objekte verwenden, ist auch diese Funktion sehr einfach. Es ist

$$f_{Objects}(T_{s_0}) = o_1 o_2 ... o_m$$

wobei für jedes $d_{ij} \in T_{s_0}$, dass in einer beliebigen Definition eines beliebigen Fluenten oder situationsunabhängigen Prädikates vorkommt, ein o_k enthalten ist. Außerdem ist für jedes d_l, dass nicht in T_{s_0}, aber in D auftaucht, ein o_k enthalten.

3. $f_{Init}(T_{s_0})$
Die Funktion $f_{Init}(T_{s_0})$ liefert eine PDDL-Kodierung des Initialzustandes zurück.
Es ist

$$f_{Init}(T_{s_0}) = (Q_1 d_{11} ... d_{1n_1})(Q_2 d_{21} ... d_{2n_2})(Q_3 d_{31} ... d_{3n_3})...$$

wobei für jeden Fluenten F_i, für den ein Ausdruck der Form $F_i(\boldsymbol{x}, s_0) \equiv ... \vee x_1 = d_{11} \wedge ... \wedge x_n = d_{1n} \vee ...$ existiert und für jedes situationsunabhängige Prädikat P_i, für das ein Ausdruck der Form $P_i(\boldsymbol{x}) \equiv ... \vee x_1 = d_{11} \wedge ... \wedge x_n = d_{1n} \vee ...$ existiert, ein Ausdruck der Form $(Q_1 d_{11} ... d_{1n})$ enthalten ist.

4. $f_{Predicates}(T_{s_0})$
Die Funktion $f_{Predicates}(D)$ liefert eine Kodierung aller vorkommenden Prädikate zurück. Da wegen E3 alle Prädikate und Fluenten einen Wert in s_0 haben müssen, kommen natürlich auch alle Prädikate und Fluenten mindestens einmal in T_{s_0} vor.
Es ist

$$f_{Predicates}(T_{s_0})) = (P_1 x_1 ... x_{n_1})(P_2 x_1 ... x_{n_2})...$$

wobei für jedes in T_{s_0} vorkommende Prädikat und jeden Fluenten ein Ausdruck der Form $(P_1 x_1 ... x_n)$ vorkommt.

5. $f_{Goal}(G(s))$
Nach Proposition 1 können wir $f_{Goal}(G(s))$ direkt in eine <GD>-Formel umschreiben. Es ist
$$f_{Goal}(G(s)) = \Xi^{PDDL}(G(s))$$

6. $f_{Actions}(D)$
Sei $N = \{n_i | n_i \text{ ist Name einer primitiven Aktion in } D\}$ die Menge aller Namen von primitiven Aktionen in D. Für jede primitive Aktion A gibt es ein Precondition-Axiom in der weiter oben festgelegten Form
$$Poss(A(x_1, ..., x_n), s) \equiv \Pi_A(x_1, ..., x_n, s)$$
Es ist
$$\begin{aligned} f_{Actions}(D) = (&: action\ n_i \\ &: parameters(h_i^{Parameters}) \\ &: precondition(\Xi^{PDDL}(\Pi_A(x_1, ..., x_n, s))) \\ &: effect(h_i^{Effect})) \end{aligned}$$
wobei

   (a) $h_i^{Parameters}$ die Parameter der Funktion angibt. Diese sind alle ungetypt und werden direkt aus dem $Poss$-Prädikat übernommen.

   (b) h_i^{Effect} die Effekte der Aktion angibt.
   Es ist
   $$h_i^{Effect} = (and\ E_1\ E_2\ ...)$$
   wobei für jedes ϕ_l, indem der Ausdruck $a = n_i$ vorkommt, ein E_i enthalten ist, dass folgendermaßen gebildet wird:

   Fallunterscheidung über die Formen (3) und (2) von ϕ_l:

      i. $(\exists ...)([\varphi_l \wedge](a = A(y_1, ..., y_m)))$
      $F(x_1, ..., x_n, do(a, s))$ ist hier wahr, falls $a = A(y_1, ..., y_m)$ und zusätzlich die Bedingung φ_l erfüllt ist. Zunächst erzeugen wir die PDDL-Syntax $\Xi^{PDDL}(\varphi_l)$ der zusätzlichen Bedingung φ_l und bilden einen konditionalen Effekt in PDDL-Syntax:
      $$(when\ \Xi^{PDDL}(\varphi_l)\ (F\ x_1\ ...\ x_n)))$$
      Da wir die Zustandsänderungen von Fluenten im Situationskalkül direkt und in PDDL über Effekte von Aktionen angeben, müssen wir die Parameter-Variablen jetzt entsprechend anpassen:
      Sei $A(v_1, ..., v_n)$ die ADL-Aktion A mit ihren Parametervariablen

v_i, $F(x_1, ..., x_m)$ das Fluent mit seinen Parametervariablen x_i und $a = A(y_1, ..., y_n)$ die parametrisierte Aktion mit den Parametern y_i. Wir ersetzen jedes x_i in $\Xi^{PDDL}(\varphi_l)$ und in $(F\ x_1\ ...\ x_n)$ durch v_j, falls $x_i = y_j$.
Schließlich wird noch über alle Variablen x_i, für die es kein y_j gibt mit $y_j = x_i$, allquantifiziert. Somit erhalten wir als Teileffekt:

$$(forall\ (x_i\ ...)\ (when\ \Xi^{PDDL}(\varphi_l)\ (F\ x_1\ ...\ x_n)))$$

ii. $F(x_1, ..., x_n, s) \wedge \neg\ ((\exists ...)([\varphi_1 \wedge](a = A_1(y_{11}, ..., y_{1m_1})))$
$\vee(\exists ...)([\varphi_2 \wedge](a = A_2(y_{21}, ..., y_{2m_2})))$
$\vee(\exists ...)([\varphi_3 \wedge](a = A_3(y_{31}, ..., y_{3m_3})))$
$\vee ..)$

hier verfahren wir für jede Aktion A_i wie unter i.; im Effekt verwenden wir zusätzlich ein *not*:
$(forall\ (X_i\ ...)\ (when\ \Xi^{PDDL}(\varphi_l)\ (not\ (F\ x_1\ ...\ x_n))))$

Beispiel 4. Gegeben sei $At(x, y, do(a, s)) = (\exists v)(a = Fahren(x, v, y))$ und die PDDL-Aktion $Fahren(lastauto\ von\ nach)$.
Wir schreiben zunächst $At(x, y)$ und ersetzen darin dann x durch *lastauto* und y durch *nach*. Da es kein x_i gibt, für das es kein y_j gibt mit $y_j = x_i$, wird nicht allquantifiziert. Es gilt außerdem $\phi = ()$, so dass kein bedingter Effekt nötig ist. So erhalten wir schließlich als Effekt E_i:
$(At\ lastauto\ nach)$

Beispiel 5. Gegeben sei $At(x, y, do(a, s)) = At(x, y, s) \wedge \neg(\exists l, n)(In(x, l) \wedge a = Fahren(l, y, n))$ und die PDDL-Aktion $Fahren(lastauto\ von\ nach)$.
Wir schreiben zunächst $At(x, y)$ und ersetzen darin dann y durch *von*. In $In(x, l)$ eretzen wir l durch *lastauto* und erhalten $(when(In\ x\ lastauto)(not(At\ x\ von)))$.
Schließlich allquantifizieren wir über x, da kein Argument von $Fahren(l, y, n))$ gleich x ist. So erhalten wir als Effekt E_i:
$(forall(x)(when(In\ x\ lastauto)(not(At\ x\ von))))$

4.3 f ist ein Kompilationsschema

Wir zeigen jetzt, dass **f** tatsächlich ein Kompilationsschema ist. Dazu müssen wir die folgenden Punkte zeigen:

K1 Es gibt einen Plan für I, gdw. es einen Plan für $F(I)$ gibt.
K2 Die Größe des Resultates von $f_{Types}, f_{Predicates}, f_{Actions}, f_{Objects}, f_{Init}$ und f_{Goal} ist polynomial in der Größe ihres Argumentes.

Wir zeigen zunächst ein Theorem, auf dem der Beweis aufbaut.

Theorem 1. *Eine eingeschränkte Basic-Action-Theorie verfügt über eine vollständige Theorie; in jeder durch einen variabelfreien Situationsterm der Form $\sigma_n = do(a_{n-1}, do(a_{n-2}, ..., do(a_1, do(a_0, s_0))))$ definierten Situation steht also für jeden Fluenten F fest, ob er wahr oder falsch ist.*

Beweis. Beweis durch Induktion über n

(IA) $\sigma_0 = s_0$
Wir haben gefordert, dass für jeden Fluenten und für jedes situationsunabhängige Prädikat eine Definition für ihren Wahrheitswert in s_0 existiert. Somit haben wir in s_0 eine vollständige Theorie.

(IV) Behauptung gelte für $\sigma_i = do(a_{i-1}, do(a_{i-2}, ..., do(a_1, do(a_0, s_0))))$

(IS) Wir haben in E2 SSA's so eingeschränkt, dass sich der Wahrheitswert eines Fluenten nur durch eine Aktion ändern kann. Desweiteren sind alle SSA's in definitionaler Form, so dass der Wahrheitswert jedes Fluenten in Situation $do(a, s)$ durch a und s deterministisch bestimmt ist. Somit folgt die Aussage direkt für $\sigma_{i+1} = do(a_i, do(a_{i-1}, ..., do(a_1, do(a_0, s_0))))$.

Definition 11. *Die wahren Fluenten einer durch einen variabelfreien Situationsterm σ definierten Situation schreiben wir als Menge $PosAtms(\sigma)$.*

Proposition 2. *Alle nicht in $PosAtms(\sigma)$ enthaltenen Atome sind falsch.*

Dies gilt offensichtlich auf Grund von Theorem 1.

Definition 12. *Wir schreiben kurz s_{i+1} für $do(a_i, do(a_{i-1}, ..., do(a_1, do(a_o, s_0))))$.*

Theorem 2. ***f** ist ein Kompilationsschema.*

Beweis.

K1 Sei $I = \langle D, T_{s_0}, G(s)\rangle$ eine BAT-Instanz. Dann ist $\mathbf{f}(I) = \langle P', P_{Init}, P_{Goal}\rangle$ die daraus erzeugte ADL-Probleminstanz in PDDL-Syntax. Sei s_0 die Initialsituation von I und z_0 der Startzustand von $\mathbf{f}(I)$.
$f_{Init}(T_{s_0})$ ist eine reine Umcodierung in PDDL-Code und verändert nichts an den Wahrheitswerten der Fluenten, so dass direkt $z_0 = PossAtms(s_0)$ folgt. Seien nun $i-1$ beliebige Aktionen angewandt, z_i sei der Zustand von $\mathbf{f}(I)$ nach Anwendung dieser Aktionen und es gelte $z_i = PossAtms(s_i)$. Wir betrachten nun einerseits den Zustand z_{i+1}, den man erreicht, wenn man a anwendet und andererseits $PossAtms(s_{i+1}) = PossAtms(do(a, s_i))$ für dieselbe Aktion a. Sei F ein Fluent. Wir betrachten nun die Wahrheitswerte von F in den Zuständen z_i und z_{i+1} bzw. den Situationen s_i und s_{i+1}. Nach Voraussetzung ist F in z_i wahr, gdw. es in s_i wahr ist.

(a) F ist in s_i wahr und in s_{i+1} falsch.
Aufgrund der Einschränkung E2 kann dies nur sein, wenn dass SSA von F ein ϕ_l enthält, das dem Schema 2 genügt und eine der Aktionen von ϕ_l die Aktion a ist. Unser Kompilationsschema enthält eine Funktion h_i^{Effect}, die dieses Schema so übersetzt, dass bei jeder Ausführung von a das Fluent F falsch gemacht wird. Also ist F auch in z_{i+1} falsch.

(b) F ist in s_i falsch und in s_{i+1} wahr.
Aufgrund der Einschränkung E2 kann dies nur sein, wenn dass SSA von F ein ϕ_l enthält, das dem Schema 3 genügt und dessen Aktion a ist. Unser Kompilationsschema enthält eine Funktion h_i^{Effect}, die dieses Schema so übersetzt, dass bei jeder Ausführung von a das Fluent F wahr gemacht wird. Also ist F auch in z_{i+1} wahr.

(c) F ist in s_i falsch und in s_{i+1} falsch.
Aufgrund der Einschränkung E2 kann dies nur sein, wenn das SSA von F kein ϕ_l enthält, dass dem Schema 3 genügt und dessen Aktion a ist. Unser Kompilationsschema setzt in so einem Fall natürlich auch keine Effekte bezüglich F für a. Da in PDDL der Wahrheitswert von Fluenten nur über Aktionen geändert wird, bleibt F auch in z_{i+1} falsch.

(d) F ist in s_i wahr und in s_{i+1} wahr.
Aufgrund der Einschränkung E2 kann dies nur sein, wenn das SSA von F kein ϕ_l enthält, dass dem Schema 2 genügt und eine Aktion a hat. Unser Kompilationsschema setzt in so einem Fall natürlich auch keine Effekte bezüglich F für a. Da in PDDL der Wahrheitswert von Fluenten nur über Aktionen geändert wird, bleibt F auch in z_{i+1} wahr.

F ist also genau dann in $\mathbf{f}(I)$ wahr, wenn F in I wahr ist. Also gilt $z_{i+1} = PossAtms(s_{i+1})$.
Falls es nun einen Plan s_{n+1} in I gibt, so gilt nach Definition 8, dass $I \models ((\forall a, s^*).do(a, s^*) \sqsubseteq s_n \supset Poss(a, s^*)) \wedge G(s_n)$. Dies heißt, dass $I \models Poss(a_i, s_i)$ für $1 \leq i \leq n-1$. Da nun wie gerade gezeigt $PosAtms(s_i) = z_i$ ist, gilt wegen Proposition 1 auch $z_i \models Pre(a_i)$.
Außerdem gilt wegen $I \models G(s_n)$ und $z_n = PosAtms(s_n)$ ebenfalls wegen Proposition 1 auch $z_n \models f_{Goal}(G(s_n))$.
Also ist $[a_1, ..., a_n]$ auch ein Plan in $\mathbf{f}(I)$.
Falls es keinen solchen Plan gibt, kann es durch eine analoge Argumentation auch keinen Plan in $\mathbf{f}(I)$ geben.
Daraus folgt die Aussage.

K2 Sieht man direkt durch die Konstruktion von **f**.

Trivialerweise bleibt die Planlänge exakt erhalten.

4.4 Ein Kompilationsschema von ADL nach eingeschränkten BAT's

Wir erzeugen aus einem ADL-Planungsproblem $P = \langle P', P_{Init}, P_{Goal} \rangle$ eine BAT-Instanz. Dazu geben wir jetzt die Funktionen $g_\Sigma, g_{T_{SSA}}(P'), g_{T_{PA}}(P'), g_{T_{UNA}}(P'), g_{T_{s_0}}(P_{Init}, P_{Objects})$ und $g_{Goal}(P_{Goal})$ an. In Kapitel 4.5 zeigen wir dann, dass es sich bei **g** um ein Kompilationsschema handelt.

1. g_Σ
   Σ sind die grundlegenden Axiome des Situationskalküls, die natürlich unabhängig von der BAT-Instanz sind. Es ist:
   $$g_\Sigma = \Sigma_1, \Sigma_2, \Sigma_3, \Sigma_4$$
2. $g_{T_{UNA}}(P')$
   T_{UNA} ist die Menge der Unique-Names-Axiome für Aktionen. Diese haben die Form $A(\boldsymbol{x}) \neq B(\boldsymbol{y})$ für je zwei unterschiedliche Aktionsnamen A und B; sowie $A(x_1, ..., x_n) = A(y_1, ..., y_n) \supset x_1 = y_1 \wedge ... \wedge x_n = y_n$ für jeden Aktionsnamen A. $g_{T_{UNA}}(P')$ erzeugt diese Axiome, von denen es $\frac{(n-1)*n}{2} + n$

viele gibt. $g_{T_{UNA}}(P')$ benötigt also nur polynomial mehr Platz als P'.

3. $g_{Goal}(P_{Goal})$
   $g_{Goal}(P_{Goal})$ erzeugt aus der ADL-Zielbedingung, die keine freien Variablen enthält, eine Situationskalkülformel mit einziger freier Variable s. Sei $P_{Init} =$ (: *goal* Φ) die Zielbedingung der ADL-Instanz. Nach Proposition 1 können wir diese als Formel in herkömmlicher prädikatenlogischer Syntax $\Xi^{FOL}(\Phi)$ schreiben. Es ist
   $$g_{Goal}(P_{Goal}) = \Xi^{FOL}(\Phi)$$
4. $g_{T_{s_0}}(P_{Init}, P_{Objects})$
   T_{s_0} enthält sowohl die Definition der Fluenten in der Initialsituaion s_0, als auch alle situationsunsabhängigen Sätze. Wir kompilieren Typdefinitionen und Objektdeklarationen in solche situationsunabhängigen Sätze.
   (a) Im ADL-Initialzustand dürfen nur positive und negative Grundatome angegeben werden. Da die Closed World Assumption besagt, dass alle nicht als wahr klassifizierten Grundatome falsch sind, genügt es, nur positive Literale anzugeben (sind doch negative angegeben, was syntaktisch zulässig wäre, können diese einfach ignoriert werden).
   $T_{s_0}^{Init}$ ist eine Menge, die alle Grundatome mit gleichem Prädikatensymbol zu jeweils einer in s_0 uniformen Definition in prädikatenlogischer Syntax zusammenfasst.

   Beispiel 6. Sei folgender ADL-Initialzustand gegeben:

   $$(: \mathit{init}\ (P_1\ A_1\ A_2)\ (P_1\ B_1\ B_2)\ (P_2\ C_1\ C_2\ C_3))$$

   Hier ist

   $$\begin{aligned} T_{s_0}^{Init} = \{\ & (P_1(x_1, x_2, s_0) \equiv (x_1 = A_1 \wedge x_2 = A_2) \vee (x_1 = B_1 \wedge x_2 = B_2)), \\ & (P_2(x_1, x_2, x_3, s_0) \equiv (x_1 = C_1 \wedge x_2 = C_2 \wedge x_3 = C_3))\} \end{aligned}$$

   (b) In ADL gibt es nur nullstellige Funktionen, diese werden hier als Objekte bezeichnet. Ein Typ definiert eine Klasse bzw. Menge von Objekten. Dabei ist es auch möglich, Hierachien zu beschreiben, wobei der Hierachiegraph für jeden Knoten nur einen Vorfahren haben darf.
   Alle verwendeten Typen werden in einer getypten Liste definiert: *(:types <typed list1(name)>)*, alle verwendeten Objekte werden in einer getypten Liste deklariert: *(:objects <typed list1(name>)*.
   Diese beiden getypten Listen haben die folgende Form:
   $\alpha_1 - \beta_1\ \alpha_2 - \beta_2 \ldots$
   wobei α_i in der Typdefinition ein oder mehrere Typen ist und in der Objektdeklaration ein oder mehrere Objekte; β_i fehlt (dann ist α_i vom Typ *Object*) oder ist wiederum ein Typ.
   Seien im folgenden $\alpha_i^T - \beta_i^T$ die Typdefinitionen und $\alpha_i^O - \beta_i^O$ die Objektdeklarationen, wobei wir davon ausgehen, dass die α_i aus jeweils nur einem Typ bzw. Objekt bestehen, statt leeren β_i nur *Object* vorkommt

und für jedes β_i^T, dass nicht als α_i^T vorkommt, ein Ausdruck $\beta_i^T - Object$ existiert (ist dieser Zustand nicht der Fall, so können wir ihn mit polynomial mehr Platzaufwand realisieren).
Sei $T := \{t_i | \alpha_i^T = t_i$ oder $\beta_i^T = t_i$ für mindestens ein α_i^T oder $\beta_i^T\}$ die Menge aller vorkommenden Typen.
$T_{s_0}^{Types}$ ist eine Menge, die für jedes $t_i \in T$ die Definition eines situationsunabhängigen Prädikates in der Form

$$t_i \equiv \alpha_1^T(x) \vee \alpha_2^T(x) \vee \alpha_3^T(x) ... \vee x = \alpha_1^O \vee x = \alpha_2^O \vee x = \alpha_3^O \vee ...$$

enthält, wobei für jedes β_j^T mit $\beta_j^T = t_i$ das passende $\alpha_j^T(x)$ und für jedes β_j^O mit $\beta_j^O = t_i$ das passende $x = \alpha_j^O$ enthalten ist. Gibt es für ein t_i keine passenden β_j^T und β_j^O, so ist $(t_i(x) \equiv \bot)$

Beispiel 7. Sei folgende ADL-Typbeschreibung gegeben:
$(: types\ Fahrrad\ Motorrad - Zweirad\ Auto)$
$(: objects\ auto - Auto\ fahrrad - Zweirad$
$motorrad - Motorrad\ bobycar)$

Dazu äquivalent mit polynomial mehr Platzaufwand ist:
$(: types\ Fahrrad - Zweirad\ Motorrad - Zweirad\ Auto - Object$
$Zweirad - Object)$
$(: objects\ auto - Auto\ fahrrad - Zweirad\ motorrad - Motorrad$
$bobycar - Object)$

Es gilt $T = \{Fahrrad, Zweirad, Motorrad, Auto, Object\}$
Somit ist

$$\begin{aligned} T_{s_0}^{Types} = \{\ & (Zweirad(x) \equiv Fahrrad(x) \vee Motorrad(x) \vee x = fahrrad), \\ & (Motorrad(x) \equiv x = motorrad), \\ & (Auto(x) \equiv x = auto), \\ & (Object(x) \equiv Auto(x) \vee Zweirad(x) \vee x = bobycar) \\ & (Fahrrad(x) \equiv \bot)\} \end{aligned}$$

Diese Definitionen entsprechen aber noch nicht der Einschränkung E3. Deshalb formen wir sie nun um; dazu definieren wir den Operator *inst*, der T_{s_0} folgendermaßen umformt:
Da wir gefordert haben, dass es keine Zyklen im Typ-Graphen gibt, gibt es mindestens einen Typen t in $T_{s_0}^{Types}$, der kein $\alpha_i^T(x)$ auf seiner rechten Seite hat. t wird in jeder Definition eingesetzt, in der er auftaucht. So wird fortgefahren, bis $T_{s_0}^{Types}$ nur noch Definitionen enthält, die alle auf der rechten Seite kein $\alpha_i^T(x)$ haben. Zum Schluß wird noch jede Definition der Form $(t_i(x) \equiv \bot)$ und jedes Disjunkt $\bot$ gestrichen.

Beispiel 8. Es ist $inst(T_{s_0}^{Types})$:

$$\begin{aligned} T_{s_0}^{Types} = \{\ & (Zweirad(x) \equiv x = motorrad \vee x = fahrrad), \\ & (Motorrad(x) \equiv x = motorrad), \\ & (Auto(x) \equiv x = auto), \\ & (Object(x) \equiv x = auto \vee x = motorrad \vee x = fahrrad \\ & \qquad \vee x = bobycar)\} \end{aligned}$$

Der Platzaufwand von $inst(T_{s_0}^{Types})$ ist polynomial in der Größe seines Argumentes.

(c) T_{s_0} muss Unique-Names-Axiome für alle Objekte enthalten. Die Menge $T_{s_0}^{UNA}$ enthält für jedes Paar von Objekten a, b aus I einen Satz der Form $a \not= b$. Die Kardinalität von $T_{s_0}^{UNA}$ ist mit der selben Begründung wie bei $g_{T_{UNA}}$ auch hier polynomial in der Anzahl der Objekte.

(d) Schließlich erzeugen wir noch ein Domain-Closure-Axiom $T_{s_0}^{DCA}$ für die eingeschränkte Basic-Action-Theorie: $\forall x(x = d_1 \vee ... \vee x = d_n)$ für alle n in $P_{Objects}$ vorkommenden Objekte.

Es ist:

$$g_{T_{s_0}}(P_{Init}, P_{Objects}) = T_{s_0}^{Init}, inst(T_{s_0}^{Types}), T_{s_0}^{UNA}, T_{s_0}^{DCA}$$

5. $g_{T_{PA}}(P')$
$g_{T_{PA}}(P')$ erzeugt die Menge T_{PA} von I, die zu jeder Aktion $A(x_1, ..., x_n)$ ein Precondition-Axiom der Form $Poss(A(x_1, ..., x_n), s) \equiv \Pi_A(x_1, ..., x_n, s)$ enthält. $\Pi_A(x_1, ..., x_n, s)$ besteht aus der in der herkömmlichen Syntax der Prädikatenlogik geschriebenen Vorbedingung $\Xi^{FOL}(\Phi)$ der ADL-Aktion A. Ein zusätzliches Konjunkt wird eingeführt, dass dafür sorgt, dass die Aktion nur für Objekte des entsprechenden Typs ausgeführt werden kann.

Beispiel 9. Sei die ADL-Aktion

$$\begin{aligned} & (:action\ A \\ & :parameters\ (p_1 - (either\ T_1\ T_2)\ p_2 - T_3) \\ & :precondition(forall(x - T_1)(and(F\ p_1\ x)(G\ p_2\ p_3)))) \end{aligned}$$

gegeben. $g_{T_{PA}}(P')$ erzeugt daraus folgendes $Poss$-Prädikat:

$$\begin{aligned} Poss(A(p_1, p_2) \equiv & ((T_1(p_1) \vee T_2(p_1)) \wedge T_3(p_2)) \\ & \wedge(\forall x(T_1(x) \supset (F(p_1, x) \wedge G(p_2, p_3)))) \end{aligned}$$

6. $g_{T_{SSA}}(P')$
Alle in der ADL-Instanz P' vorkommenden Fluenten werden in einer (: *predicates*)-Klausel deklariert. $g_{T_{SSA}}(P')$ erzeugt für jeden dieser Fluenten ein Successor State Axiom. Dabei wird jeder Effekt einer ADL-Aktion, der diesen Fluenten wahr oder falsch macht, in das SSA „eingebaut“, indem in

einem ϕ_l des SSA die Aktion zusammen mit eventuellen Bedingungen notiert wird. Dabei wird jeweils ein Ausdruck der Form 3 für wahrmachende und ein Ausdruck der Form 2 für alle falschmachenden Aktionen gewählt.
Getypte Aktions-Parameter werden wie bei $g_{T_{PA}}$ als einstellige Prädikate kompiliert und als Konjunkt im SSA kodiert.
Für jeden Fluenten $(F\ v_1 - T_1\ ...\ v_n - T_n)$ erzeugt $g_{T_{SSA}}(P')$ also ein SSA:

$$\begin{aligned} F(v_1, ..., v_n, do(a, s)) \equiv\ & (T_1(v_1) \wedge ... \wedge T_n(v_n)) \\ & \wedge(\xi_1 \vee \xi_2 \vee \xi_3 \vee ... \\ & \vee(F(v_1, ..., v_n, s) \wedge \neg(\zeta_1 \vee \zeta_2 \vee \zeta_3 \vee ...)) \end{aligned}$$

Dabei werden die ξ_l und ζ_l für die einzelnen SSA's erzeugt, indem der Effekt E jeder PDDL-Aktion $(A\ x_1\ x_2\ x_3\ ...)$ solange zerlegt wird, bis die beeinflussten Prädikate erreicht sind. Dann kann dieser Teil des Effektes einem SSA zugeordnet werden.
Während der Zerlegung erhält jeder der auftretenden Teileffekte E' eine ihm zugeordnete Menge $\Phi_{E'}$, die zusätzliche Bedingungen enthält.
Die Zerlegung erfolgt rekursiv und unterscheidet zwischen folgenden Fällen:

(a) Falls E von der Form $(and\ A\ B\ C\ ...)$, zerlegen wir A, B und C separat und es gilt $\Phi_A = \Phi_B = \Phi_C = \Phi_{(and\ A\ B\ C\ ...)}$.

(b) Falls E von der Form $(forall\ (x - (either\ T1\ T2...)...)\ E')$, zerlegen wir E' weiter und es gilt $\Phi_{E'} = \Phi_E \cup ((T_1(x) \vee T_2(x)) \wedge ...)$.

(c) Falls E von der Form $(when\ \Pi\ E')$, zerlegen wir E' weiter und es gilt $\Phi_{E'} = \Phi_E \cup \Xi^{FOL}(\Pi)$.

(d) Falls E von der Form $(P\ y_1\ y_2\ ...\ y_n)$, so brechen wir diesen Zweig der Zerlegung ab und bilden eine Formel $(a = A(x_1, x_2, x_3...)) \wedge \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge ...$ für alle $\phi_i \in \Phi_E$.
Dann müssen wir für noch die Variablen anpassen. Sei $F(v_1, v_2, ..., do(a, s))$ das Fluent mit den Parametern, die es im SSA hat. Wir gehen für $1 \leq i \leq n$ folgendermaßen vor:
   i. falls $x_i = y_j$, ersetze x_i durch v_j
   ii. falls $y_i \neq x_j$ für alle j, ersetze y_i durch v_i in allen ϕ_i. Jetzt existenzquantifizieren wir noch über alle übriggebliebenen x_i und erhalten so schließlich ein ξ_l von P.

$$((\exists ...)(a = A(x_1, x_2, x_3...)) \wedge \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge ...)$$

(e) Falls E von der Form $(not\ (P\ y_1\ y_2\ ...\ y_n))$, verfahren wir analog wie unter (d) erhalten hierbei dann jeweils ein ζ_l von P.

Beispiel 10. Sei $A = (:\ action\ Fahren\ :\ parameters\ (auto\ von\ nach)...)$ mit Effekt $E=$

```
:effect (and (At auto nach)
             (not (At auto von))
             (forall (x - (either Mann Frau))
                     (when (In x auto)
```

```
                              (and (At x nach)
                                   (not (At x von))
                              )
                         )
                    )
               )
```

E ist offensichtlich von der Form (a); wir zerlegen also im Weiteren `(At auto nach)`, `(not (At auto von))` und

```
(forall (x - (either Mann Frau))
        (when (In x auto)
              (and (At x nach)
                   (not (At x von))
              )
        )
)
```

getrennt.

Offensichtlich ist E' =`(At auto nach)` von der Form (e), es ist $\Phi_{E'} = \emptyset$, also schreiben wir $a = fahren(auto, von, nach...)$. Jetzt ersetzen wir *auto* durch v_1 und *nach* durch v_2. Da wir *von* nicht ersetzen können, existenzquantifizieren wir darüber und erhalten mit

$$\exists von(a = fahren(v_1, von, v_2))$$

ξ_1 von $At(v_1, v_2)$.

`(not (At auto von))` behandeln wir auf analoge Weise und erhalten mit

$$\exists nach(a = fahren(v_1, v_2, nach)$$

ζ_1 von $At(v_1, v_2)$.
Der letzte Teil E'' ist von der Form (b), deshalb gilt $\Phi_{E''} = \{(Mann(x) \vee Frau(x))\}$ und wir zerlegen:

```
(when (In x auto)
      (and (At x nach)
           (not (At x von))
      )
)
```

Dieser Teil E''' nun ist von der Form (c), deshalb gilt $\Phi_{E'''} = \{(Mann(x) \vee Frau(x)), In(x, auto)\}$ und wir zerlegen `(and (At x nach) (not (At x von))))` weiter. Dies ist wieder von der Form (a) und wir zerlegen nun `(At x nach)` und `(not (At x von))` einzeln.

Für `(At x nach)` bilden wir $a = fahren(auto, von, nach) \wedge (Mann(x) \vee Frau(x)) \wedge$

$In(x, auto)$ und ersetzen noch *nach* durch v_2 und in $(Mann(x) \vee Frau(x)) \wedge In(x, auto)$ noch x durch v_1. *auto* und *von* konnten wir nicht ersetzen, also existenzquantifizieren wir darüber. So erhalten wir ξ_2 von $At(v_1, v_2)$:

$$\exists auto, von(a = fahren(auto, von, v_2) \wedge (Mann(v_1) \vee Frau(v_1)) \wedge In(v_1, auto))$$

`(not (At x von))` wird genauso behandelt und liefert ζ_2:

$$\begin{aligned}\exists auto, nach(\ & a = fahren(auto, v_2, nach) \\ & \wedge (Mann(v_1 \vee Frau(v_1)) \wedge In(v_1, auto)\end{aligned}$$

Unter der Annahme, dass $Fahren$ die einzige Aktion ist, die $At(x, y)$ beeinflusst, und dass alle Parameter ungetypt sind, lautet das SSA von $At(x, y)$ also:

$$At(v_1, v_2, do(a, s)) \equiv \xi_1 \vee \xi_2 \vee (At(v_1, v_2, s) \wedge \neg(\zeta_1 \vee \zeta_2))$$

4.5 g ist ein Kompilationsschema

Wir zeigen jetzt, dass **g** tatsächlich ein Kompilationsschema ist. Dazu müssen wir die folgenden Punkte zeigen:

K3 Es gibt einen Plan für P, gdw. es einen Plan für $G(P)$ gibt.
K4 Die Größe des Resultates von $g_\Sigma, g_{T_{SSA}}, g_{T_{PA}}, g_{T_{UNA}}, g_{T_{s_0}}$ und g_{Goal} ist polynomial in der Größe ihres Argumentes.

Theorem 3. ***g** ist ein Kompilationsschema.*

Beweis. K3 Sei $P = \langle P', P_{Objects}, P_{Init}, P_{Goal}\rangle$ eine ADL-Probleminstanz in PDDL-Syntax. Dann ist $\mathbf{g}(P) = \langle D, T_{s_0}, G(s)\rangle$ die daraus erzeugte BAT-Instanz. Sei z_0 der Startzustand von P und s_0 die Initialsituation von $\mathbf{g}(P)$. $g_{T_{s_0}}$ formt die in z_0 wahren Fluenten nur syntaktisch um, so dass direkt $PossAtms(s_0) = z_0$ folgt. Seien nun $i-1$ beliebige Aktionen angewandt, z_i sei der Zustand von P nach Anwendung dieser Aktionen und es gelte $PossAtms(s_i) = z_i$. Wir betrachten nun einerseits den Zustand z_{i+1}, den man erreicht, wenn man a anwendet und andererseits $PossAtms(s_{i+1}) = PossAtms(do(a, s_i))$ für dieselbe Aktion a. Sei F ein Fluent. Wir betrachten nun die Wahrheitswerte von F in den Zuständen z_i und z_{i+1} bzw. den Situationen s_i und s_{i+1}. Nach Voraussetzung ist F in z_i wahr, gdw. es in s_i wahr ist.

(a) F ist in z_i wahr und in z_{i+1} falsch.
Dies kann nur sein, wenn der Effekt der Aktion a den Fluenten F falsch macht. In diesem Fall sorgt aber $g_{T_{SSA}}$ dafür, dass im einzigen Schema der Form 2 die Aktion a vorkommt, so dass dieses nicht erfüllt sein kann. Natürlich kann a in keinem anderen ϕ_l vorkommen, deshalb ist F auch in $do(a, s_i)$ falsch.

(b) F ist in z_i falsch und in z_{i+1} wahr.
Dies kann nur sein, wenn der Effekt der Aktion a den Fluenten F wahr macht. In diesem Fall sorgt aber $g_{T_{SSA}}$ dafür, dass in einem Schema der Form 3 die Aktion a vorkommt, so dass dieses erfüllt ist und damit auch F in $do(a, s_i)$ wahr wird.

(c) F ist in z_i falsch und in z_{i+1} falsch.
Entweder hat a keinen Effekt auf F, dann gibt es auch kein Schema der Form 3 im SSA von F, indem a vorkommt und F bleibt in $do(a, s)$ falsch, oder a macht F falsch, dann kommt die Aktion a im einzigen Schema der Form 3 des SSA von F vor und wird somit auch in $do(a, s_i)$ falsch.

(d) F ist in z_i wahr und in z_{i+1} wahr.
Entweder hat a keinen Effekt auf F, dann kommt a auch nicht im Schema der Form 2 im SSA von F vor und F bleibt in $do(a, s)$ wahr, oder a macht F wahr, dann gibt es ein Schema der Form 3 im SSA von F und somit wird F auch in $do(a, s_i)$ wahr.

F ist also genau dann in $\mathbf{g}(P)$ wahr, wenn F in P wahr ist. Also gilt $z_{i+1} = PossAtms(s_{i+1})$.
Falls es nun einen Plan $[a_1, ..., a_n]$ in P gibt, so gilt nach Definition 7, dass es eine Sequenz von Zuständen $\{z_i\}_{i=0...k+1}$ gibt, so dass a_i in z_i anwendbar ist, z_0 der Initial-Zustand der Planungsinstanz ist und für jedes $i = 0...k$ gilt, dass z_{i+1} aus z_i entsteht, indem alle Grundatome aus Add_a zu z_i hinzugefügt werden und alle Grundatome aus Del_a aus z_i gelöscht werden. Wenden wir die selben Aktionen in $\mathbf{g}(P)$ an, sind, wie eben gezeigt, in den Situationen $\{s_i\}_{i=0...k+1}$ die selben Grundatome wahr und, da $g_{T_{PA}}$ nur syntaktisch umformt, gilt auch $\mathbf{g}(P) \models ((\forall a, s^*).do(a, s^*) \sqsubseteq \sigma \supset Poss(a, s^*))$. Außerdem gilt wegen $z_{i+1} \models G$ auch $\mathbf{g}(P) \models G(s)$. Also ist $[a_1, ..., a_n]$ auch in $\mathbf{g}(P)$ ein Plan.
Falls es keinen solchen Plan gibt, kann es duch eine analoge Argumentation auch keinen Plan in in $\mathbf{g}(P)$ geben.
Daraus folgt die Aussage.

K4 Sieht man direkt durch die Konstruktion von $\mathbf{g}$.

5 Zusammenfassung und Schlußbemerkungen

Wir haben das ADL-Fragment von PDDL (PDDL 2.1 Level 1) und den Situationskalkül, auf dem GOLOG basiert, auf ihre Ausdrucksmächtigkeit hin untersucht. Nachdem wir festgestellt hatten, das der Situationskalkül sehr viel mächtiger als ADL ist, haben wir sogenannte *eingeschränkte* Basic-Action-Theorien definiert, die über dieselbe Ausdrucksmächtigkeit wie ADL verfügen.
Um zu zeigen, dass Basic-Action-Theorien, die diesen Einschränkungen genügen, tatsächlich nicht mehr mächtiger sind, haben wir ein Kompilationsschema von eingeschränkten BAT's nach ADL angegeben. Dass nun aber nicht etwa ADL mächtiger ist, als es eingeschränkte BAT's sind, haben wir durch ein weiteres Kompilationsschema von ADL in eingeschränkte BAT's gezeigt.
Aufbauend auf diesen Ergebnissen kann man nun sowohl die Situationskalkül-Seite als auch die PDDL-Seite um eine ihrer zahlreichen Erweiterungen ergänzen und die so entstandenen Teilsprachen auf ihre Mächtigkeit hin untersuchen. Dazu kann man entweder wiederum Kompilationsschemata angeben oder aber zeigen, dass eine Teilsprache ausdrucksstärker ist als die andere, indem man beweist, dass kein passendes Kompilationsschema existieren kann.

Literatur

1. Fox, M., Long, D.: An extension to pddl for expressing temporal planning domains. Journal of Artificial Intelligence Research (2003)
2. Reiter, R.: Knowledge in Action. MIT Press (2001)
3. Nebel, B.: On the compilability and expressive power of propositional planning formalisms. Journal of Artificial Intelligence research (2000)
4. Green, C.: Application of theorem proving to problem solving. Proceedings of the 1st International Joint Conference on Artificial Intelligence, pages 219-240 (1996)
5. Fikes, R.E., Nilsson, N.J.: Strips: a new approach to the application of theorem proving to problem solving. Proceedings of the Australian Joint Conference on Artificial Intelligence, 2:189-208 (1971)
6. Pednault, E.P.D.: ADL: Exploring the middle ground between STRIPS and the situation calculus. Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (1989)
7. McDermott, D.: PDDL - The Planning Domain Definition Language, Version 1.2. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control (1998)
8. Lifschitz, E.: On the Semantics of STRIPS. Proceedings of 1986 Workshop: Reasoning about actions and plans (1986)
9. Levesque, H., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.: GOLOG: A logic programming language for dynamic domains. Journal of Logic Programming, 31:59-84 (1997)
10. Gerevini, A., Long, D.: Plan Constraints and Preferences in PDDL3. Technical Report, Department of Electronics for Automation, University of Brescia, Italy (2005)